ICASE

LANGUAGE CONCEPTS FOR DISTRIBUTED PROCESSING

OF LARGE ARRAYS

Piyush Mehrotra

and

Terrence W. Pratt

Report No. 82-14

June 16, 1982

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING

NASA Langley Research Center, Hampton, Virginia 23665

Operated by the

UNIVERSITIES SPACE USRA RESEARCH ASSOCIATION

LANGUAGE CONCEPTS FOR DISTRIBUTED PROCESSING OF LARGE ARRAYS

Piyush Mehrotra
University of Virginia

and

Terrence W. Pratt
University of Virginia

Abstract

A large array is an array whose storage is distributed among primary and secondary storage and whose processing may be distributed among several tasks in a distributed system. This paper presents a semantic model (set of language concepts) for representing large arrays in a distributed system in such a way that the performance realities inherent in the distributed storage and processing can be adequately represented. An implementation of the large array concept as an ADA package (abstract data type) is described, as well as a particular tailoring of the concept for the NASA Finite Element Machine. An example application program using the package is also described.

This work supported in part by NSF Grant MCS78-00763 and NASA Contract NAS1-46. Partial support was also obtained under NASA Contract Nos. NAS1-14472 and NAS1-16394 while the authors were in residence at the Institute for Computer Applications in Science and Engineering (ICASE), NASA Langley Research Center, Hampton, Virginia.

1. Introduction

Distributed processing of large data structures is an important problem that has received relatively little attention at the level of programming language design. This paper presents a set of language concepts, that is, a semantic model, to support the processing of large arrays on a variety of distributed and parallel computer architectures. The semantic model is intended to present a unified view to the user of large arrays, so that the user may structure his program as an array processing program, without radical distortion of the underlying algorithm. However, the arrays in question are understood to be partitioned and distributed among processor memories and secondary storage in a distributed architecture. This means that the user view cannot simply be the traditional language view of arrays, because there are performance realities associated with the distribution and movement of portions of the array during distributed processing that should not be masked from the user. As Jones and Schwarz [4] note in their study of use of the CM*, performance realities in a distributed computation cannot realistically be masked entirely from the user without a major degradation in performance of a distributed system. This paper presents the semantic model, described initially as an ADA package defining an abstract data type "large array". An implementation design for an MIMD distributed computer, the NASA Finite Element Machine [5], is described and an application for this machine is sketched.

2. Background

Most large scientific computations involve intensive processing of large arrays. The potential for computational speedup by distributing array processing in various ways among multiple processors has been the driving force behind the design of most vector and SIMD supercomputers such as the ILLIAC-IV, the CRAY 1, the CDC CYBER 205, and the NASA Massively Parallel Processor (MPP) [2]. MIMD distributed systems provide an opportunity for taking advantage of the parallelism inherent in these computations in different ways. The NASA Finite Element Machine (FEM) [5], a 36 processor distributed system, and the CMU CM^* represent two radically different MIMD architectures that might potentially be used in such applications.

Language structures proposed for array processing on parallel computers have primarily concentrated on extension of ordinary language arrays to include specification of parallelism in certain directions, such as is done in Perrott's ACTUS [9]. Language proposals for distributed computation, such as CSP [3], ARGUS [6], and ADA have not treated the problem of large data structure processing (beyond consideration of ordinary file processing).

There are two coupled problems in the distributed processing of large data structures such as arrays for which we seek an effective language treatment:

1. The partitioning of an array and its distributed processing on the separate processors of an MIMD system. We seek a solution that allows large scale concurrent processing of shared data structures without a major overhead in task communication or unnecessary mutual exclusion as tasks traverse the data structures. Guardians [6] and monitors, for example, provide mutual exclusion but sharply limit the concurrency available, while semaphores, critical regions, and rendezvous allow more concurrency but only with a large communication overhead.

2. The partitioning of an array and its distributed storage in both secondary storage and processor memories in an MIMD system. To process a large data structure such as an array usually requires a complex series of data partitionings and data movements through a distributed system. It is this problem that presents a major difficulty even for users of conventional vector and SIMD array computers. For example, Perrott and Stevenson [10] report that users of the ILLIAC-IV found data partitioning and data movement to be the most difficult aspect of programming for that machine. For distributed storage, we seek a solution that frees the user from the task of managing directly the complexities of storage and data movement inherent in the use of a distributed system, without masking the critical performance realities involved.

The roots of our approach are found more directly in traditional language structures for sequential file processing than in those for array processing. Consider the PASCAL view of files. A file is a large linear data structure in PASCAL. It might easily be considered as a one-dimensional array rather than as a separate data type (for economy of language concept), but Wirth avoided this temptation in the PASCAL design, as most other language designers have done. The fact that PASCAL arrays are of invariant size while files may be extended at one end is the least important difference between files and vectors (APL, for example, allows new elements to be concatenated to a vector). Ignoring this difference, there appear to be three main distinctions between files and vectors. Each is a linear sequence of components of arbitrary type in PASCAL, but (1) a file is a large data structure presumed to be distributed between secondary and central memory, while a vector is resident entirely in central memory, (2) processing of a file at random points is not possible, due to the distributed nature of its storage, but instead is restricted to a window which makes only part of the file visible at one time, and (3) a file has a lifetime that is potentially <u>longer</u> than that of the program processing it, so that its structure is defined independently of the program processing it.

The language concept of file processing is straightforward. A program wishing to process a file is given a window (one component wide in PASCAL) on the file. Processing is only possible within the window, but the window may be moved in certain regular patterns on the file. To process an entire file, a program positions its window at one end, and then alternates processing and window moving steps until the entire file has been traversed. An end-of-file function allows the end of the file to be detected. To extend the file, the window is positioned just past the end of the file, and a new component is assigned.

Implementation of file processing is also an area where distributed processing concepts have been widely used on a limited scale. Typically, two processors cooperate, one executing the user program and the other managing the buffering of blocks of data from secondary storage into buffers in central memory. From there the first process moves the data into user program variables (representing its processing "window") for processing as demanded by the program. The user is effectively protected from having to manage these transfers himself, but the language concepts of "window" and "moving the window" reflect more abstractly the performance realities inherent in the implementation structure.

3. Language Concepts for Large Array Processing

In many applications that involve processing of large arrays, the arrays are stored and processed in a manner more similar to that appropriate for file processing than for array processing in traditional languages. That is, the arrays are large data structures that must be stored at least partially on secondary storage, their lifetime is different from that of the programs processing them, they are processed in blocks that effectively represent a processing window on the array, and the pattern of processing involves a regular (and often repeated) traversal of the entire structure by alternating steps of processing and moving. We wish to provide a language semantic model to support this view of arrays and array processing, and extend it appropriately for a distributed computation on each array.

The semantic model is based on the following concepts:

1. A large array is to be seen by the programmer as a single data structure with the same logical organization as an ordinary array. For example, a large matrix (two-dimensional array) is organized as a grid of rows and columns in the usual way. This allows the use of algorithms developed for matrix processing to be used without radical distortion of their structure.

2. A large array, however, is not ordinarily visible to a single task as a unit at one time. Instead a task sees only a part of the array through a rectangular window. Only the part of the array visible in the window may be accessed and modified. Thus the window is the locus of processing for the task.

3. A window may be positioned on an array by a task and subsequently moved as needed. Thus to process an entire array, a window is created and positioned on the array. The data visible in the window is processed, and then the window is moved to a neighboring position. Processing and moving

alternate until the array is completely traversed. A regular movement pattern may be expressed in a task using an extension of a FOR loop (a CLU type iterator mechanism [7] is appropriate). Alternatively, the MOVE operation may be directly invoked as required.

4. A task may subdivide its window into smaller parts called subwindows. Subwindows may be passed as parameters to subtasks for concurrent processing. The subtasks may synchronize and communicate in the usual ways to exchange information during processing (e.g., to request values from a neighboring subwindow). However, subwindows cannot be moved independently; only the entire window may be moved. Thus the processing proceeds in phases. The window is moved by the main task. The subtasks are invoked to process the data within their subwindows. When all the subtasks have terminated, the main task may again move the window. Partitioning of a window into subwindows is done statically as part of the window definition rather than dynamically.

5. Windows and subwindows may be created with three different types of access privileges: Read-only, write-only, and read-write. Overlap of read-only subwindows is allowed, but write-only and read-write subwindows must be disjoint.

6. Operations are provided to allow a task to detect the "borders" of an array (analogous to the usual end-of-file test).

7. A task may have several windows on different arrays, and these may be moved asynchronously as needed.

8. Several tasks may have windows concurrently on the same array, but write-only and read-write windows are not allowed to overlap.

This conceptual model for large array processing allows the language structure to reflect the performance realities of distributed storage and processing without unduly burdening the user with implementation details. Thus the user may be made aware that a MOVE is costly and inhibits concurrent processing (since all subtasks must terminate) without losing the conceptual unity of viewing the data structure as an array (rather than as separate blocks distributed between secondary and primary storage).

4. "Large Array" as an Abstract Data Type

The class of large arrays is appropriately considered as a new abstract data type. As such it might be included as an extension to an existing language or as part of a new language for distributed computing. To gain some experience with use and implementation of the semantic model outlined above, we have chosen to define the model as an abstract type within ADA, using the generic package facility. The ADA tasking facility provides a "virtual computer" that may be considered as an abstract distributed machine. Within this abstract machine, the LARGE_ARRAY package defines a detailed implementation of the large array model described above, and also provides a syntax (not entirely ideal) for creating and operating on large arrays.

An important aspect of the LARGE_ARRAY abstract type is that it subsumes the basic model for ordinary sequential file processing (exclusive of special features for text files). Thus a one-dimensional large array (vector) represents a generalized form of sequential file, in which the window may be larger than a single element. If two tasks each have a window simultaneously on a single large vector, such that one is generating new elements while the other is processing existing elements, the model becomes that of the UNIX pipe, which is also an extension of the usual sequential file processing concept to a (more restricted) distributed system.

The public specification part of the ADA generic package is given in Appendix A. Space does not permit the complete definition to be given, so its basic internal structure is described informally here.

Creating large arrays

As can be seen from the package definition, one or more large arrays containing elements of the same type can be supported by one instantiation of the generic package via the type LARGE_ARRAY. Preexisting arrays in external files can be accessed by attaching one of these large arrays via the procedure OPEN, while new arrays can be created using the procedure CREATE. The row and column bounds of a large array are specified at the time of associating the large array with an external file and remain fixed during the existence of the array. Procedure CLOSE can be used to sever the association of an internal large array with its associated external file. Procedure DELETE deletes the associated external file.

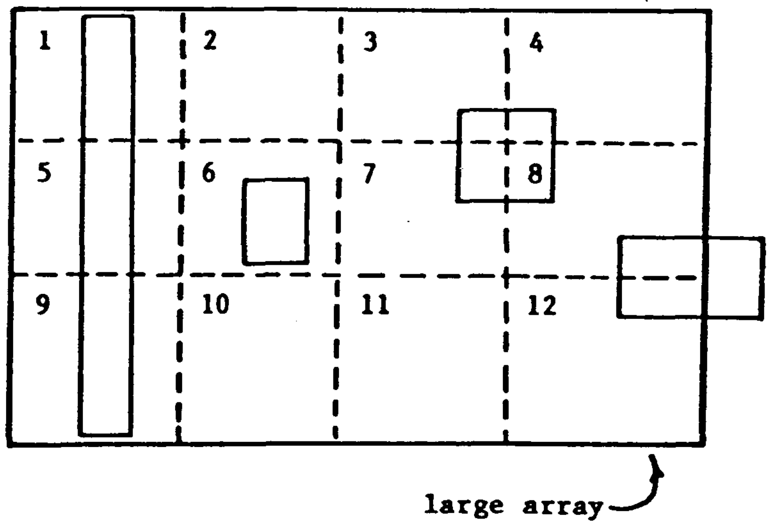
The package internally views a large array as a sequence of blocks, each block being defined as a subarray of the large array as shown in Fig. 1. The block is used as a unit for transfer of data between secondary storage and primary memory(s). Thus the block size is implementation dependent and determined by the optimum size for I/O data transfer.

The row and column bounds are used to determine the total number of blocks needed for the large array. In our ADA implementation, for each of the blocks constituting the large array a monitor-like task of type BLOCK CONTROLLER is initiated to control access to the block. The reading and writing of the block to secondary storage is performed by the associated task as and when required. These BLOCK_CONTROLLER tasks are discussed further below in conjunction with the implementation of window movement.

Windows

Windows are statically attached to a particular large array using the procedure CREATE. The same procedure is also used to specify the size of the window and its privileges, i.e., whether it is read-only (R), write-only (W), or read-write (RW). The row and column increments to be used for relative movement of the window are also passed as parameters to the procedure. The user can specify an edge element that will be used to fill out the portion of the window that does not lie within the bounds of the associated large array when the window is moved past the edge of the array.

A window is viewed by the user as an array of specified size along with information which is private to the package. Thus the processing of the elements of the large array visible through the window is performed in a manner analogous to



------indicates block boundary
————indicates window boundary

Each numbered block has an associated BLOCK_CONTROLLER task.

Fig. 1. Large array with four windows, showing the division into blocks in the ADA implementation.

the processing of an ordinary small array. That is, the accessing of window elements is done through subscripting relative to the origin of the window rather than the origin of the large array.

Moving a window

Two methods of window movement are provided: relative and absolute. The MOVE procedure uses the row and column increments (defined when creating the window) to move the window to a new position relative to the present position. The SET procedure, on the other hand, moves the window to the indicated absolute position on the array. Thus SET can be used to establish the initial position of the window on the array.

Whenever a window is moved, a copy of the elements visible through the window is provided. When the window is moved again, the values of the elements are written back on the array before the move is executed. Thus each process works on a private copy of the elements visible through the window, and the actual array is ordinarily updated only when the window is moved away. Procedures READ and WRITE are available to update the window or the associated large array respectively without actually moving the window.

Depending upon the size of the window and the position of the window within the large array, the window may partially or fully cover one or more blocks of the array, as shown in Fig. 1. Internally for each window a list of blocks covered by the window is maintained. The procedure READ sequentially makes an entry call to each of the BLOCK_CONTROLLER tasks associated with the blocks in this list in order to update the appropriate portion of the window. Similarly, WRITE makes an entry call to the tasks for updating the appropriate portion of the blocks.

When a window is to be moved to a new position several steps need to be performed. First the elements at the present position are updated (for write-only and read-write windows) with calls to the tasks controlling the blocks covered by the window. Then entry calls are made to these tasks to detach the window from the blocks.

Next the blocks that would be covered by the window in the new position are determined. Entry calls are made to the tasks controlling these blocks to attach the window to the blocks. Each task determines if the window can overlay its block without causing an incompatible overlap with a window already stationed on the block (since write-only and read-write windows cannot overlap). If such a overlap would occur, the window to be moved is delayed until the other window has moved off the position. Otherwise the BLOCK_CONTROLLER task attaches the window to the block. Once attach entry calls for all the blocks covered by the window have been successfully completed, the elements at the new position are read into the window (local array), thus completing the operation of moving a window.

Block controller tasks

Each of the BLOCK_CONTROLLER tasks controlling a block of the array maintains a list of the windows attached to the block. Since a window may be only covering a part of the block, information about the portion of the block covered by the window and about the corresponding part of the window which overlays the block, is also maintained by the task for each of the windows attached to it. Thus when a task accepts a READ or WRITE call for a particular window, it can determine the appropriate portions of the window and the block that are to be used for the purpose of reading or writing.

A task controlling a block of the large array has four entry points (a) attach a window, (b) detach a window, (c) read and (d) write. For an attach entry call, the task checks for incompatible overlap with windows already on its window list. If no overlap can occur or if one of the overlapping windows is a read-only window then the window is attached to the window-list along with the relevant data regarding the portion of the block being overlayed by the window and the part of the window covering the block. Whenever a window is attached to an empty window-list, the task controlling the block reads the block in from secondary storage. Thus the block is resident in memory only while a window is covering it.

For a detach entry call, the particular window is detached from the window-list. If this causes the window-list to become empty the block is written back onto secondary storage. The transfer to secondary storage is performed only if the block has been modified while it was resident in memory. In either case, the memory required for the block is then released.

For a read entry call, the appropriate portion of the block is copied into the window. Similarly for a write entry call, the copying is done from the window into the block.

Mutual exclusion and deadlock

The use of an independent task controlling access to each block of a large array in the ADA implementation provides mutual exclusion of external tasks wishing to access the block in order to move their windows. Once a copy of the relevant portion of the block has been made into a window, independent external tasks can process their windows concurrently. The use of tasks for controlling the blocks thus provides a monitor-like mutual exclusion, but it does not sharply limit the possible concurrency available for processing of the arrays. Note that direct access to BLOCK_CONTROLLER tasks is not available to the user because these tasks are hidden within the LARGE_ARRAY package.

Deadlock when several windows are moving asynchronously on the same array is avoided by a simple resource ordering strategy. Blocks are numbered sequentially and become the "resources" that must be requested in a fixed sequence. A MOVE or SET operation must proceed by first detaching the window from each block it covers and then attaching the window to the blocks in its new position in the order of the block numbers. No reading or writing of portions of a window may occur until the window has been successfully attached to all the blocks it covers.

Subwindows

Subwindows can be statically overlayed on a window via the procedure CREATE provided for this purpose. The window with which the subwindow is to be associated, its position within the window and its privileges are passed as parameters. The privileges of a subwindow have to be compatible with those of the associated window (e.g., a write-only window cannot have a read-only subwindow). Also the subwindow configuration overlaying a window should not cause an incompatible overlap between subwindows with different privileges.

Procedure ASSIGN is used to assign a value to a particular element of a subwindow. Function GET returns the value of a specified element. These two subprograms work directly on the window to which the subwindow is attached (rather than on a copy). Note that there are no procedures for moving subwindows; only the window on which the subwindow is overlayed can be moved. A move automatically updates the elements in the subwindow.

A function EOS has been provided for both a window and a subwindow to determine if the window or subwindow is straddling the edge of the large array structure. This function returns an integer signifying the particular edge or corner that has been reached, or zero if the window or subwindow is entirely within the boundaries of the array. Various other functions are provided to determine the properties of large arrays, windows, and subwindows.

The semantics of the large array model has been described above in terms of the LARGE_ARRAY package. A more complete description may be found in [8]. The package has been implemented on the VAX 11/780 at the University of Virginia using the UNIX implementation of the NYU ADA/ED translator and interpreter.

5. Implementation on the Finite Element Machine

Implementation strategies for large array processing are another aspect of this project. Rather than fix entirely the semantic details of the general large array model and then search for an effective implementation strategy on different distributed architectures, we have chosen instead to explore different tailorings of the general semantic model to fit particular architectures.

The distributed system to which we have immediate access is the NASA Finite Element Machine (FEM), currently running in a four processor version at the NASA Langley Research Center, with a 16 processor version expected to be complete by the end of 1982 and a 36 processor version available sometime in 1983. The FEM is an MIMD architecture originally designed for finite element calculations in structural engineering. However the architecture is general purpose except for minor details. A TI 990 minicomputer serves as a controller and interface with secondary storage and the external environment. Each of the 36 FEM processors is a standard TI 9900 microcomputer with separate local memory. There is no shared memory. The processors are arranged in the form of a square array, and each can communicate directly with its eight nearest neighbors or over a global bus to any other processor or the controller. The local communication lines for the processors on the "logical edge" of the array wrap around. A separate network composed of a set of boolean "signal flags" is provided for synchronization of the processors.

Currently for the programmer on the FEM (using Pascal), if a large array is to be processed, the programmer must partition the array, store it on secondary storage in blocks, and explicitly code the transfers of the blocks through the controller to the individual processors as needed. Adams and Ortega [1] describe a typical finite element calculation of this sort.

The implementation design for the large array model on the FEM is organized as follows. A large array is stored on a secondary storage device in blocks or pages. The controller memory is used as a "staging buffer" (somewhat like a buffer in ordinary file I/O). The controller handles all requests for movement of windows. Thus all the data structures needed to implement windows and subwindows are maintained by the controller itself.

The controller receives requests for creating and opening large arrays and associates large arrays with external files. The creation of windows and subwindows is also performed by the controller. Each time a task executes the procedure CREATE for defining a window or a subwindow, the controller is signaled and the relevant data is sent to it. Along with the data pertaining to the size, position, etc., of the windows and subwindows, the controller also stores information about the task or subtask (and the processor) with which each is associated.

When a task requests that a window be positioned at a particular place on the large array using the SET procedure, the request is sent to the controller, which brings the relevant pages from secondary storage into its local memory, and then transfers the data to the processor memories. When a task requests that its window be moved (by a call to the MOVE procedure), the request is sent to the controller, which uploads the relevant blocks from the FEM processors, modifies the pages in its "staging buffer", writes these pages out to secondary storage, brings in the new pages required, and reloads the processor memories with the new data. This is the process that now must be done manually by the programmer. Similarly a READ or WRITE request for a window results in downloading or uploading of data to the processor memories by the controller. Note that in the FEM implementation there is no need for the BLOCK_CONTROLLER tasks used in the ADA implementation since all requests for block access are handled sequentially by a single task in the controller.

Each subwindow is associated with a subtask on one of the FEM processors. When subwindows are overlayed on a window, storage for each of the subwindows of the window is allocated within each subtask. When the main task positions the window on the large array, the data is downloaded on the global bus to the subwindow area within each subtask. The subtasks can synchronize themselves using the hardware supported flag network after processing of the data within the subwindows has been completed. At this point, the main task can again move the window (which results in uploading the subwindow data to the staging buffer in the controller and the downloading of new data to the subwindow areas in the processors). The subtasks can then be restarted to process the new subwindow data.

In the general semantic model for large arrays, overlapping subwindows may be used to allow one subtask to access data that is being processed by another subtask. Implementation of this structure without shared memory on the FEM is difficult. On the FEM subtasks resident on different processors instead may communicate directly using the local and global communication links. Overlapping subwindows are then unnecessary and may be prohibited.

The main task that "owns" a window and controls its movement may be resident either on the controller or on one of the processors. It may have a subwindow itself where it processes the array, or it may only serve as a synchronizer to monitor the activity of the subtasks and move the window when each processing step has been completed.

To match the hardware realities of the FEM, we have made several restrictions on the general large array model:

a. A window must be either completely partitioned into non-overlapping subwindows or it must be entirely without subwindows.

b. Subwindows are associated with subtasks statically (at the time of their creation) rather than being passed as parameters to the subtasks.

6. Example

The Finite Element Machine has been designed to solve finite element problems in structural analysis. One of the major phases in the finite element method of analyzing structures is the solution of a system of simultaneous equations of the form:

$$A\ x = b$$

In this section, an algorithm for solving such a system of equations, where the matrix A is an n by n upper triangular matrix, is sketched. The algorithm essentially uses a direct back solve method for solving the equations for a set of right hand side vectors b. The algorithm has been coded in ADA using the LARGE_ARRAY package (see Appendix B), but assuming the restrictions of the FEM implementation rather than the general ADA implementation. A sketch of the algorithm of Appendix B is given here; see [8] for this and other complete algorithms.

For the purposes of this algorithm the FEM would be configured as a linear sequence of n processors, each communicating only with its left and right neighbors. The wrap around feature of the FEM would be used to connect processors on the edge. For the purposes of coding the algorithm, ADA task entry calls have been used to simulate the FEM local neighbor connections.

Each of the first n-1 processors would have a subtask of type BACK_SOLVE resident upon it. The subtask MAIN_BACK_SOLVE would be resident on the nth processor (see App. B). The algorithm is set up so that each of the processors views one column of the array A and performs partial calculations on the solution vector x. Thus the subtask MAIN_BACK_SOLVE "receives" the right hand side vector b and calculates the nth element of x, calculates the contribution of the nth column of the array A and passes on the partially calculated x vector to its left neighbor. The kth subtask BACK_SOLVE receives the partially calculated x vector from its right neighbor, calculates the kth element of the x vector, and updates the rest of the x vector.

The procedure MAIN resident on the controller declares A, X, and B as LARGE_ARRAY's and OPEN's A and B, associating them with already existing external files. A new file is created and associated with the large array X. A read-only window, A_window, is also defined by MAIN and associated with the large array A. A_window is defined to be as large as the entire array A and is positioned so as to cover the whole array.

Each of the BACK_SOLVE subtasks declares a subwindow, A_sw, of the window A_window and positions it such that the subwindow for subtask k is at the kth column of the window. Also a one element window is declared by each subtask on the large array X and is initially positioned on the first column of the kth row of X.

The subtask MAIN_BACK_SOLVE, in addition to the A_sw subwindow and the X window, also has a window, B_window, on the large array B. This window is initially positioned at the first column of the array B. After processing this first column of the B_window, the subtask calculates the nth element of the x vector, writes it in the X_window, passes its contribution to its left neighbor (subtask n-1) and then moves the B_window and X_window one position to the right on the respective arrays so as to process the next b vector. Processing and movement alternate until the end of the large array B is reached (EOS(B_window) is negative), indicating that solution for all the given right sides (in B) is complete.

The kth subtask BACK_SOLVE receives k elements of the partially calculated x values from its right neighbor. It generates the kth element of the x vector in X_window, communicates k-1 partially calculated x values to its left neighbor, and then moves its X_window. This is repeated until all right hand sides have been processed and all subtasks are at their TERMINATE SELECT alternative, at which time they all terminate at once.

7. Conclusions

The large array model is intended to provide a conceptual unity at a high-level to an area of distributed processing that is now treated only with relatively low level primitives. The central concepts of window and subwindow allow both the distributed storage and the distributed processing of a large array to be represented in an applications program in a way that is natural for array processing but which at the same time reflects the performance realities involved.

Implementation of the large array package on different distributed systems would take on a different form in each case. We have chosen to first define a general semantic model and then to study the tailoring of the model that is appropriate for different distributed architectures.

There is still a great deal to be learned about the representation and implementation of large data structures and data structure processing on distributed systems. Within the confines of this particular model, several areas of importance are:

a. Dynamic extension of arrays. It is natural to consider tasks as extending arrays in various ways during processing (continuing the analogy with file processing in traditional languages). Although we have touched on these issues here to a limited extent, a more complete semantics and implementation model for such dynamic extension is needed.

b. Dynamic windows. The windows discussed in this model are conceptually static as far as their size and other parameters such as move increments are concerned. The semantics and implementation for more dynamic windows need to be explored.

c. Special types of arrays. The package supports only matrices in the form given here. The extension to higher dimension arrays appears straightforward but needs to be explored, as well as special types of arrays such as symmetric, sparse, and banded matrices.

REFERENCES

[1] Adams, L. and Ortega, J., "A Multi-color SOR Method for Parallel Computation," Proc. 1982 Intl. Conf. on Parallel Processing, August 1982.

[2] Batcher, K. E., "Design of a Massively Parallel Processor," IEEE Trans. Comput., C-29, No. 9, September 1980, pp. 836-840.

[3] Hoare, C. A. R., "Communicating Sequential Processes," CACM, Vol. 21, No. 8, August 1978, pp. 666-677.

[4] Jones, A. and Schwarz, P., "Experience Using Multiprocessor Systems - A Status Report," ACM Comp. Surveys, Vol. 12, No. 3, June 1980, pp. 121-166.

[5] Jordan, H. F., "A Special Purpose Architecture for Finite Element Analysis," Proc. 1978 Intl. Conference on Parallel Processing, August 1980, pp. 263-266.

[6] Liskov, B. and Scheifler, R., "Guardians and Actions: Linguistic Support for Robust, Distributed Programs," Ninth ACM POPL, Albuquerque, NM, January 1982, pp. 7-19.

[7] Liskov, B., et al., "Abstraction Mechanisms in CLU," CACM, Vol. 20, No. 8, August 1977, pp. 564-576.

[8] Mehrotra, P. "Distributed processing of large arrays", Ph.D. thesis, University of Virginia, 1982 (in preparation).

[9] Perrott, R., "A Language for Array and Vector Processors," ACM TOPLAS, Vol. 1, No. 2, October 1979, pp. 177-195.

[10] Perrott, R. H. and Stevenson, D. K., "Users' Experiences with the ILLIAC IV System and its Programming Languages," SIGPLAN Notices, Vol. 16, No. 7, July 1981.

Appendix A. ADA Generic Package for Large Arrays (specification part only).

```
generic
    type ELEMENT is private;
package LARGE_ARRAY_PKGE is
    type LARGE_ARRAY is private;
    type MATRIX is array(INTEGER range <>, INTEGER range <>) of ELEMENT;
    type WINDOW_INFO is private;
    type WINDOW_DESC(row_size, col_size: NATURAL) is
            record
                win:    MATRIX(1..row_size, 1..col_size);
                info:   WINDOW_INFO;
            end record;
    type WINDOW is access WINDOW_DESC;
    type SUBWINDOW is private;
    type PRIVILEGES is (R,W,RW);

-- Procedures to create or open large arrays (attach internal names to external large arrays).
    procedure CREATE(   ar: in out LARGE_ARRAY;
                        row_low_bd, row_high_bd, col_low_bd, col_high_bd: INTEGER;
                        name: in STRING);
    procedure OPEN(     ar: in out LARGE_ARRAY;
                        row_low_bd, row_high_bd, col_low_bd, col_high_bd: INTEGER;
                        name: in STRING);

-- Procedures to delete or close large arrays.
    procedure DELETE(   ar: in out LARGE_ARRAY);
    procedure CLOSE(    ar: in out LARGE_ARRAY);

-- Procedures for creating windows and subwindows.
    procedure CREATE(   wind: in out WINDOW;
                        row_size, col_size: NATURAL;
                        inmode: PRIVILEGES;
                        row_inc, col_inc: INTEGER;
                        ar: LARGE_ARRAY;
                        edge: BOOLEAN;
                        edge_element: ELEMENT);
    procedure CREATE(   subwin: in out SUBWINDOW;
                        row_size, col_size, row_pos, col_pos: NATURAL;
                        inmode: PRIVILEGES;
                        wind: WINDOW);

-- Procedures to move windows.  Note that a move implies a write of the last window position (if the
-- window is not read-only) and a read of the next position (if the window is not write-only).
    procedure SET( wind: in out WINDOW;              -- absolute movement
                   new_row, new_col: INTEGER);
    procedure MOVE( wind: in out WINDOW);            -- relative movement

-- Procedures to read and write windows without movement of the window.
    procedure READ(     wind: in out WINDOW);
    procedure WRITE(    wind: in WINDOW);

-- Procedures to assign and get values of single elements of subwindows.
    procedure ASSIGN(   subwin: in out SUBWINDOW;
                        row, col: NATURAL;
                        value: ELEMENT);
    function GET(       subwin: SUBWINDOW;
                        row, col: NATURAL) return ELEMENT;

-- Functions to determine the end of structure.
    function EOS(   wind: in WINDOW) return INTEGER;
    function EOS(   subwin: in SUBWINDOW) return INTEGER;

-- Various other functions are included in the package but not listed here that
-- return the various properties of large arrays, windows, and subwindows.

end LARGE_ARRAY_PKGE;
```

Appendix B. ADA Program for Parallel Solution of a Set of Simultaneous Equations.

```
with LARGE_ARRAY_PKGE;
procedure MAIN is
    package FLOAT_ARRAY is new LARGE_ARRAY_PKGE(float);
    use FLOAT_ARRAY;

    n: constant NATURAL;          -- size of the A matrix
    m: constant NATURAL;          -- number of right hand sides to be solved

    task type BACK_SOLVE is
        entry WHO_AM_I(self_id: NATURAL);
        entry MORE;
        entry NEXT(x: float);
    end BACK_SOLVE;
    task body BACK_SOLVE is separate;

    task MAIN_BACK_SOLVE is
    end MAIN_BACK_SOLVE;
    task body MAIN_BACK_SOLVE is separate;

    SOLVE: array (1..n-1) of BACK_SOLVE;
    A,X,B: LARGE_ARRAY;
    A_window: WINDOW;

    begin
        OPEN(A,1,n,1,n,"A_file");       -- Large array A is n x n; its external file name is A_file.
        OPEN(B,1,n,1,m,"B_file");       -- Large array B is n x m; its external file name is B_file.
        OPEN(X,1,n,1,m,"X_file");       -- Large array X is n x m; its external file name is X_file.
        CREATE(A_window,n,n,R,0,0,A,FALSE);       -- Create A_window as an n x n window on A;
                                                  -- A_window never moves, so its move increments are 0.
        SET(A_window,1,1);                        -- Position A_window to cover all of A.
        for i in 1..n-1 loop
            SOLVE(i).WHO_AM_I(NATURAL(i));        -- Inform each of the SOLVE tasks of its id number.
        end loop;
    end MAIN;


separate (MAIN)
task body MAIN_BACK_SOLVE is

    B_window, X_window: WINDOW;
    A_sw: SUBWINDOW;

    begin
        CREATE(A_sw,n,1,1,n,R,A_window);          -- Create A_sw as an n x 1 read-only subwindow on the
                                                  -- nth column of A_window.
        CREATE(X_window,1,1,W,0,1,X,FALSE);       -- Create X_window as a 1 x 1 write-only window on X;
                                                  -- set the move increments to (0,1), i.e., move along
                                                  -- a row, one column at a time.
        SET(X_window,n,1);                        -- Position X_window at row n, column 1 on X.
        CREATE(B_window,n,1,R,0,1,B,FALSE);       -- Create B_window as an n x 1 read-only window on B;
                                                  -- set the move increments to (0,1), as noted above.
        SET(B_window,1,1);                        -- Position B_window at row 1, column 1 of B.
    solve_cycle:
        loop
            X-window(1,1) := B_window(n,1) / A_sw(n,1);     -- Calculate the nth element of the x vector
                                                            -- and write it in the X_window.
            SOLVE(n-1).MORE;                                -- Signal the left neighbor, subtask n-1,
                                                            -- that more is to be done.
            for i in 1..n-1 loop       -- Pass the partially calculated x values on to the left neighbor.
                SOLVE(n-1).NEXT(B_window(i,1) - A_sw(i,1) * X_window(1,1));
            end loop;
            MOVE(B_window);            -- Move the B_window one column to the right.
            MOVE(X_window);            -- Move the X_window one column to the right.
            if EOS(B_window) /= 0 then
                exit solve_cycle       -- Exit if processing of right sides is complete, i.e.,
            end if;                    -- if B_window is outside of the B array.
        end loop solve_cycle;
    end MAIN_BACK_SOLVE;
```

```
separate (MAIN)
task body BACK_SOLVE is

    X_window: WINDOW;
    A_sw: SUBWINDOW;
    id: NATURAL;
    partial_x: array(1..n) of float;

    begin
        accept WHO_AM_I(self_id: NATURAL) do      -- Find out your own id.
            id := self-id;
        end WHO_AM_I;
        CREATE(A_sw,n,1,1,id,R,A_window);         -- Create A_sw as an n x 1 read-only subwindow
                                                  -- on the id'th column of A_window.
        CREATE(X_window,1,1,W,0,1,X,FALSE);       -- Create X_window as a 1 x 1 write-only window on X;
                                                  -- set the move increments to (0,1), i.e., move along
                                                  -- a row, one column at a time.
        SET(X_window,id,1);                       -- Position X_window at the id'th row, 1st column of X.
        loop
            select
                accept MORE do end MORE;          -- Wait for more processing signal.
                    for i in 1..id loop           -- Accept id partial x vector values from right neighbor.
                        accept NEXT(x: float) do
                            partial_x(i) := x;
                        end NEXT;
                    end loop;
                    X_window(1,1) := partial_x(id) / A_sw(id,1);     -- Calculate the id'th element of the
                                                                     -- x vector and write it in X_window.
                    SOLVE(id-1).MORE;             -- Signal left neighbor subtask that more work is ready.
                    for i in 1..id-1 loop         -- Pass partial x vector values to left neighbor.
                        SOLVE(id-1).NEXT(partial_x(i) - A_sw(i,1) * X_window(1,1));
                    end loop;
                    MOVE(X-window);               -- Move X_window one column to the right.
            or
                terminate;                        -- Terminate when all subtasks are waiting for more work.
            end select;
        end loop;
    end BACK_SOLVE;
```